A Deep Learning Based Cost Model for Automatic Code Optimization

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Enabling compilers to automatically optimize code has been a longstanding goal for the compiler community. Efficiently solving this problem requires using precise cost models. These models predict whether applying a sequence of code transformations reduces the execution time of the program. Building an analytical cost model to do so is hard in modern x86 architectures due to the complexity of the microarchitecture. In this paper, we present a novel deep learning based cost model for automatic code optimization. This model was integrated in a search method and implemented in the TIRAMISU compiler to select the best code transformations. The input of the proposed model is a set of simple features representing the unoptimized code and a sequence of code transformations. The model predicts the speedup expected when the code transformations are applied. Unlike previous models, the proposed one works on full programs and does not rely on any heavy feature engineering. The proposed model has only 16% of mean absolute percentage error in predicting speedups on full programs. The proposed model enables TIRAMISU to automatically find code transformations that match or are better than state-of-the-art compilers without requiring the same level of heavy feature engineering required by those compilers.

1 Introduction

Writing high-performance software is essential in many areas from machine learning to science and engineering. In nuclear physics, for example, researchers need to perform large scale simulations to study the properties of matter. A highly optimized implementation of these simulations can be orders of magnitude faster compared to an unoptimized implementation. In deep learning, an optimized implementation of a state-of-the-art neural network such as XLNet [16] is $1.8\times$ faster than the equivalent PyTorch implementation. Writing such a highly optimized code requires ninja programmers and is time-consuming while the results are error-prone, less understandable, and non-portable. One of the longstanding goals in the compiler community is to develop compilers that can automatically optimize high-level code. These compilers automatically apply code transformations to make the code run faster; thus, avoiding the need for manual low-level program tuning. They provide greater productivity, portability, and high performance, and will be directly accessible by domain scientists.

Automatically generating efficient code for high-performance systems is a tedious task. In order for the compiler to generate efficient code, two problems have to be solved. First, a large set of critical code transformations and a mechanism to apply them to programs need to be provided. Examples of such transformations include loop fission, fusion, parallelization, and vectorization. Second, the right sequence of code transformations from this large set has to be chosen. The selected code transformations must preserve the program semantics and provide the highest performance for the input program. While state-of-the-art-compilers have shown success in solving the first problem (i.e., the ability to provide a large set of transformations and correctly apply a selected sequence of transformations [15, 5, 13, 7, 8, 11]), they still do not successfully solve the second problem (i.e., selecting the sequence of transformations that will provide the best performance).

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

The problem of selecting the right sequence of code transformations can be modeled as a search problem that can be solved in three steps. In the first step, the compiler uses a search technique to explore the space of possible code transformations. The result of this step is a set of candidates where each one is a sequence of code transformations. In the second step, the compiler checks the validity of each candidate (i.e., checks that applying the transformations does not change the program semantics). In the third step, the compiler evaluates the valid candidates and chooses the one that minimizes the execution time. This evaluation can be done by running each candidate on the target hardware to obtain the exact speedup. However, this is not a feasible solution in practice as running a program takes a considerable amount of time. Moreover, the exact hardware may not be available at compile time. Another way to evaluate a candidate is by using a cost model to predict the speedup.

Designing cost models manually is known to be a hard task [14, 2]. This is mainly due to the diversity of hardware architectures and their complexity (out-of-order execution, complex memory hierarchies, data prefetching, etc.). Complex interactions between code transformations make the problem more complicated. Recently, cost models, such as Ithemal [10] and Halide [1], have demonstrated how to overcome some of this complexity by using deep learning. While these state-of-the-art cost models are more accurate, they are limited in two ways: Ithemal [10] only predicts throughput for basic blocks of assembly code (instead of full programs). It also assumes that data is always in cache. The cost model in Halide [1] requires heavy feature engineering (it uses 54 complex program features). Designing such features is tedious, error-prone, and time-consuming.

In this paper, *we propose a novel DNN-based cost model* that avoids the problems of previous work. Our model operates on full programs expressed in a high-level language (not just basic blocks). It takes into consideration not only memory accesses to the cache but also to the main memory. Moreover, it does not require heavy feature engineering. The proposed cost model takes the original unoptimized code and a sequence of code transformations and predicts the speedup that these transformations would yield when applied. The model is designed for CPUs and is integrated in the TIRAMISU compiler [4], a compiler for the TIRAMISU domain-specific language (DSL). Because this model is a regression model, it allows the compiler to select the best transformation candidates by ranking the candidates selected by a search technique.

Contributions In summary, the contributions of this paper are:

- A novel deep-learning-based cost model for code optimization. This cost model is a *regression* cost model, operates on *full programs*, and *does not rely on extracting complex features*.
- A training data set that includes 1.8 million automatically generated programs.
- An implementation of the proposed model and an integration into a search approach to enable the TIRAMISU compiler to automatically search for the best code transformations.
- We evaluate the proposed model and show that it has a low error rate reaching 16% mean absolute percentage error. We show also that it enables TIRAMISU to automatically find code transformations that match or outperform state-of-the-art compilers.

2 TIRAMISU Embedded DSL

TIRAMISU [4] is a domain-specific language (DSL) embedded in C++. It provides a C++ API that allows users to write a high level, architecture-independent algorithm, and a set of API calls to select which code transformations should be applied. The first part of a TIRAMISU program specifies the algorithm without specifying how it should be optimized. The second part specifies which code transformations to apply and how the results of computations should be stored. This is similar to the Halide language [12], except that TIRAMISU provides additional program analysis and code transformations as it uses a mathematical model known as the polyhedral model internally [6, 5, 3, 4]. The following code shows an example of a convolution algorithm written in TIRAMISU.

```
// Declare the iterators.
var n(0, batch), fout(0, out_features), fin(0, in_features), y(0, H-2), x(0, W-2),
     k0(0, 3), k1(0, 3);
// Algorithm.
conv(n, fout, y, x) += weights(fout, fin, y, x) * input(n, fin, y + k0, x + k1);
```

The iterators in line 2 define the loop bounds around the `conv` computation. The algorithm is semantically equivalent to the following code.

```
for (n in 0..batch)
 for (fout in 0..out_features)
  for (y in 0..H-2)
   for (x in 0..W-2)
    for (fin in 0..in_features)
     for (k0 in 0..3)
      for (k1 in 0..3)
       conv[n, fout, y, x] += weigths[fout, fin, y, x] * input[n, fin, y+k0, x+k1];
```

The next code shows an example of code transformation commands that can be applied to the previous convolution kernel. These commands apply parallelization, loop interchange, tiling, vectorization, and unrolling.

```
// Provide the code transformation commands.
conv.parallelize(n);
conv.interchange(fout, fin);
conv.tile(y, x, 32, 32);
conv.vectorize(fout, 8);
conv.unroll(k0); conv.unroll(k1);
```

Currently, in TIRAMISU, a developer has to provide the previous sequence of code transformations manually. Our goal is to automate finding that sequence. We do this by developing a cost model that predicts the speedup of using a given transformation or any sequence of valid transformations. For example, the model can be used to predict whether combining parallelization, loop interchange, and loop tiling is useful. In addition, the model can be used to choose the right arguments for each one of the previous code transformations (e.g., choose the tile sizes).

3 Data Generation

As training DNNs requires a large data set and only a small number of programs have ever been written in TIRAMISU, we decided to automatically generate a data set and use it to train the model. We developed a code generator that generates random programs and sequences of code transformations. Each one of these randomly generated programs and code transformations is compiled, executed, and finally, the actual speedup is measured. The speedup is the ratio between the execution time of the original unoptimized program and the optimized one. Each data point in the data set is a triplet of the form (program, a sequence of code transformations, measured speedup).

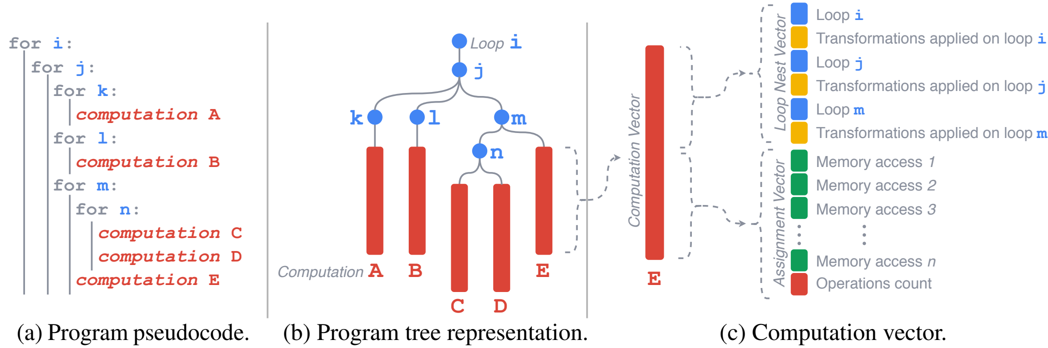
Random code generation A TIRAMISU program is a sequence of computations where each computation is an assignment. There are three common patterns of assignments that appear in TIRAMISU programs: (1) simple assignments where the right-hand side is a function of input arrays or array values computed previously; (2) stencils (e.g. horizontal blur); (3) reductions (e.g. matrix multiplication). The random code generator generates sequences of computations where each computation is a variant (or a combination) of the previous patterns. Randomly generated programs are correct by construction. A computation consumes either constants, input arrays, or values computed by previous computations. Code transformations are also generated randomly, but specific rules are used to guarantee that code transformations are valid (for example, tiling is not applied if the loop extent is smaller than the tile size).

4 Program Characterization and Model Architectures

Our cost model is designed to support programs that can be expressed in TIRAMISU. The latter is designed for expressing data parallel algorithms that operate over dense arrays using loop nests and sequences of statements. These algorithms are often found in image processing, deep learning, dense linear algebra, tensor operations, and stencil computations. A formal description of programs supported by TIRAMISU can be found in [4]. Code transformations supported by the proposed model, currently, include loop fusion, loop tiling, loop interchange, and loop unrolling which are all challenging. For simpler transformations such as parallelization and vectorization, we use simple heuristics [12].

4.1 Program characterization

Designing complex hand-engineered features is tedious, error-prone, and time-consuming. Instead of using complex hand-engineered features, we characterize programs by extracting simple high-level information that is stored in a compact variable-size representation.

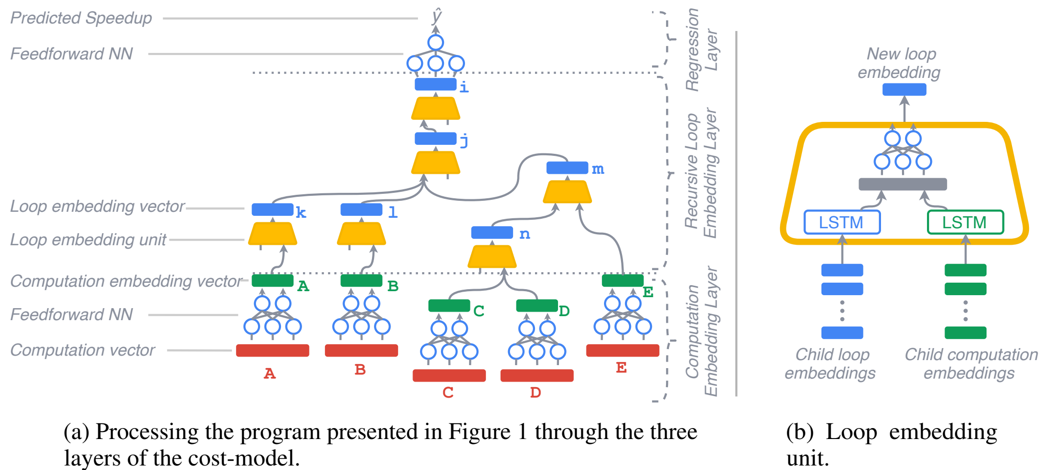


(a) Program pseudocode. (b) Program tree representation. (c) Computation vector.

Figure 1: Our characterization of a typical program.

Our program characterization is based on the AST (Abstract Syntax Tree) representation of programs. A program is characterized as an ordered tree of *computation vectors* as shown in Figure 1b. A *computation vector* is a vector that includes three pieces of information: *loop nest representation*, *assignments representation* and *loop transformation representation*. We use a tree structure to encode the *program structure*.

4.2 Model Architecture



(a) Processing the program presented in Figure 1 through the three layers of the cost-model. (b) Loop embedding unit.

Figure 2: The cost model architecture

We model the problem of speedup estimation as a regression problem: given an algorithm and a set of code transformations, our model predicts the speedup expected when applying the suggested code transformations compared to the base program (i.e. without applying code transformations).

We design our cost model's architecture to support the variable size and recursive nature of our program characterization by combining Recurrent and Recursive Neural Networks. Our model's architecture has three layers as shown in Figure 2a.

5 Search Space Exploration

Finding the best code transformations is a hard combinatorial optimization problem due to the fact that constraints, i.e. interaction between code transformations, and the objective, i.e. the speedup, cannot be mathematically represented using the program's features. Thus, the proposed model is used as an objective function estimator to better navigate the search space. However, the used search exploration approach should take into account the estimator's margin of error, thus requiring stochasticity in the search space exploration.

Since constraints cannot be related to each other, one of the best ways to model the problem of finding the best code transformations and their parameters is to use a tree search. This allows us to

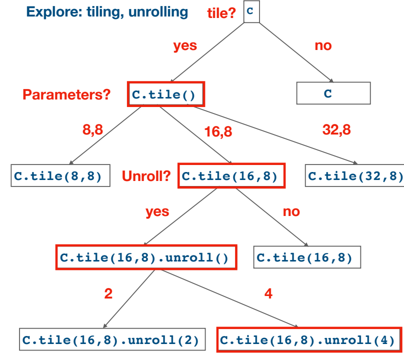


Figure 3: Example of the BS Tree for exploring the tiling and unrolling code transformations

use classical tree search algorithms. In this paper, we use Beam Search and MCTS (Monte Carlo Tree Search).

The Beam Search tree (as shown in Figure 3) explores whether to apply a code transformation and which parameters to use for that transformation. At each node of the tree, an evaluation is conducted using the cost model to assess whether the chosen transformations provide a good speedup. In Figure 3, exploring the tree shows that applying tiling with a tile size of (16, 8) and unrolling with a factor of 4 provides the best sequence of code transformations.

6 Evaluation

To evaluate our cost model: (1) we measure its accuracy on a test set composed of random programs and compare the predicted and the measured speedups on that data set; (2) we measure the speedups obtained when the model is used to search for code transformations in real-world benchmarks; (3) we compare the accuracy of this model with the accuracy of the model used in Halide [12], a state-of-the-art model.

The model evaluation and the data collection are performed on 16 identical multi-core CPU nodes. Each node has a dual-socket, each socket is a 12-core Intel Xeon E5-2680v3 CPU, with 128 GB RAM. We used 60% of data for training, 20% for validation, and 20% for testing.

$$MAPE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

Model Accuracy To measure the accuracy of the proposed model, we use MAPE (Mean Absolute Percentage Error), where y and $\hat{y}$ are respectively the measured and the predicted speedups. The MAPE of our cost model on the test set is 16%.

The Pearson correlation coefficient for the proposed model is 0.90, showing that the linear correlation between predicted and measured speedups is strong. In addition, we evaluate the ranking capabilities of the model with the Spearman's rank correlation coefficient, defined as: $r_s(y, \hat{y}) = r\big(rg(y), rg(\hat{y})\big)$ where $rg(y)$ converts the speedups to ranks and r is the Pearson correlation coefficient. The Spearman's rank coefficient of our cost model is 0.95, which shows that the predicted and measured ranks are highly linearly correlated. This property is important when using the model with a search method.

Comparing Predicted and Measured Speedups Figure 4 compares the predicted and measured speedups. To simplify visualization, we use a subset of the test set. This subset is composed of 100 random programs, each with 32 random sequences of code transformations (therefore, the total is 3200 transformed programs). The horizontal axis is the list of 3200 programs. These programs are sorted based on their speedups in ascending order to simplify visualization. As the figure shows, the predicted speedups are close to the measured ones. The error in prediction is lower around the speedup 1 and is higher as the speedup gets further from 1. We will comment more on this behavior later in the section.

Figure 5 investigates the distribution of the model error rates over the whole test set. On top, Absolute Percentage Error (APE) is measured on the code transformations of each program and the results are plotted through a histogram. On bottom, APE is measured on all data points of the test set and the measured speedups are plotted against their APE. We can see that the error gets smaller as speedups approach 1 and gets higher as speedups get far from 1. Particularly, the error is more significant for speedups below 0.05. The model is more accurate around speedup 1 because most programs in the training data set have speedups close to 1. Speedups below 0.05

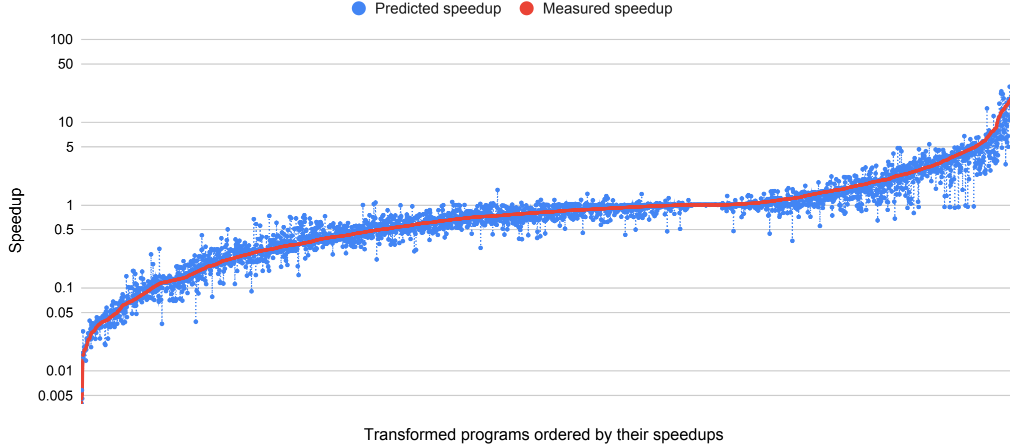


Figure 4: Predicted speedups compared to measured speedups. The speedups are ordered in ascending order.

are less frequent. The next experiment will evaluate whether the accuracy of the model allows finding the best code transformations when searching the space.

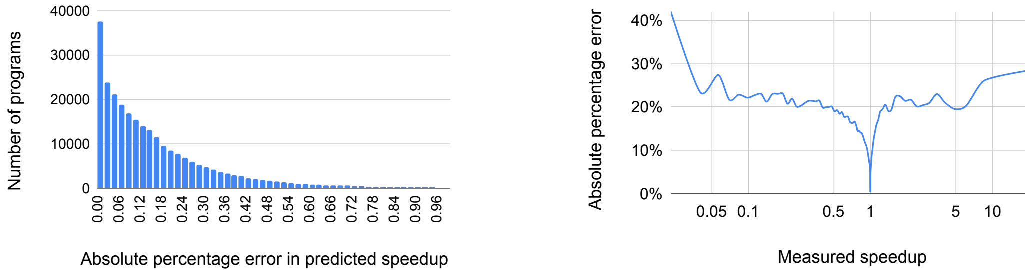


Figure 5: The distribution of error rates for the whole test set. On top, APE is measured for each transformed program, then the histogram of measurements is plotted. On bottom, APE is measured for each transformed program, then the speedups are plotted with their APE.

Search Space Exploration Using the Cost Model In this experiment, we evaluate the ability of search approach combined with the cost model to find good code transformation sequences for real-world benchmarks. We use BS and MCTS to explore the search space. We use a set of real-world benchmarks spanning different areas: image processing, deep learning, linear algebra and stencils. The benchmarks include *box blur* (an image processing filter to blur images), *conv + relu* (two successive neural network layers that benefit from operator fusion), *convolution* (a direct neural network convolution), *cvtcolor* (an image processing filter for converting the colors of an input image from RGB to gray), *doitgen* (a kernel from the multiresolution adaptive numerical scientific simulation [9]), *heat2d* (heat equation over 2D space), *heat3d* (heat equation over 3D space), *jacobi2d* (a jacobi-style stencil computation over 2D data with 5-point stencil pattern), *mvt* (matrix vector multiplication composed with another matrix vector multiplication but with transposed matrix), and *seidel2d* (Gauss-Seidel style stencil computation over 2D data with 9-point stencil pattern). The sizes of the input data for each benchmark is provided in appendix.

Figure 6 shows the best speedups found for each benchmark. The baseline is the original program where the outermost loop is parallelized (no other code transformation is applied). The first column (blue), reports results obtained when beam search is used to explore the search space. This column is considered the reference in our comparison as execution is used to obtain the speedups. In the second and third columns, beam search and MCTS use the cost model to predict speedups. The last column shows the speedups obtained after applying the Halide autoscheduler (Halide automatic optimizer) defined in [1].

Beam search with the cost model is competitive in most benchmarks, but does not find the best code transformations in *heat2d*, *jacobi2d* and *seidel2d*. Beam search with the cost model relies entirely on predictions to make decisions. Bad predictions can thus mislead the search method which is why beam search does not find the best transformations in the previous benchmarks. MCTS has similar performance, except in *jacobi2d* and *seidel2d* where it finds better code transformations, and in *cvtcolor* where the code transformations found are less good. MCTS can find better code transformations in these cases because it copes with model imprecision taking into account its stochasticity. However, since the tree space is explored differently, MCTS might explore different nodes compared to BS and thus have distinguishable results.

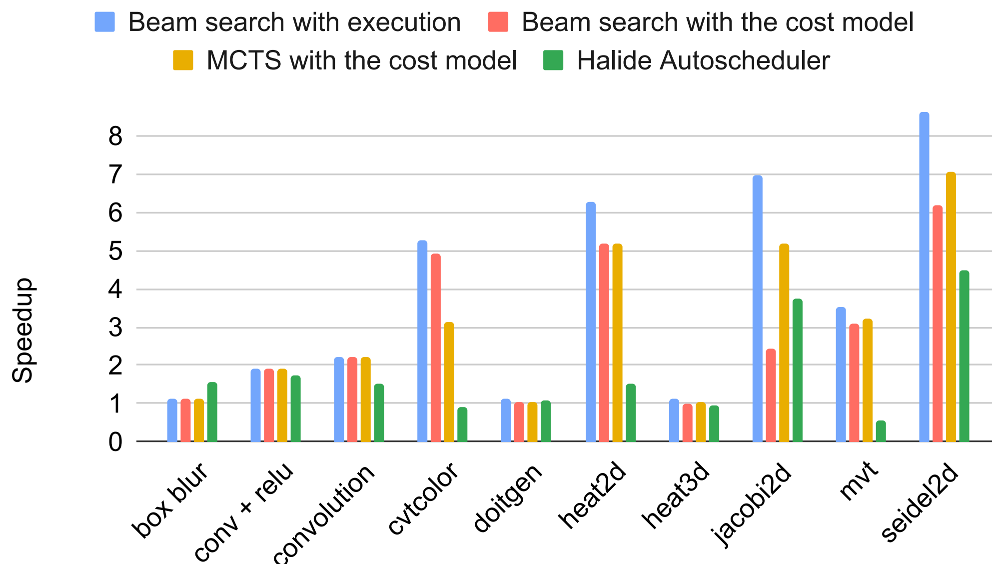


Figure 6: Speedups for different benchmarks obtained by exploring the search space.

Comparison with Halide In this section, we compare our cost model with the one of Halide [1], a state-of-the-art cost model and the closest to ours. In comparison with Halide, TIRAMISU finds transformation sequences that are either competitive with those found by Halide or better (except in *box blur*). This is mainly due to miss predictions by the Halide model which lead Halide to use transformations that degrade performance. These wrong predictions happen in particular in benchmarks that are from the area of scientific computing which Halide was not trained to handle (*heat2d*, *jacobi2d*, *mvt* and *seidel2d*). In benchmarks that fall in the categories of deep learning and image processing, which Halide supports well, TIRAMISU and Halide have comparable performance.

We also compare the performance of the Halide model with that of TIRAMISU on randomly generated programs. Halide's paper uses R^2 as an accuracy metric and uses MSE (Mean Square Error) as a loss function, we thus use the same metric and loss function for comparison. Halide has an R^2 of 0.96, whereas TIRAMISU has 0.89. Both Halide and TIRAMISU have comparable results but Halide uses heavy feature engineering. The main advantage of TIRAMISU is that it does not require feature engineering.

7 Conclusion

This paper presents a novel cost model for predicting speedups. This cost model is a *regression* cost model that operates on *full programs* and *does not rely on extracting complex features*. It is not limited to transformation parameters but also includes code transformations. We develop a random code generator to generate the training data and release the generator and the data publicly. We evaluated the proposed model and show that it had a low error rate of 16% MAPE. We integrate this model in a search space method and show that the integrated approach enables TIRAMISU to automatically find sequences of code transformations that are competitive with state of the art compilers.

References

[1] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.

[2] M. Bachir, F. Brault, D. Gregg, A. Cohen, et al. Minimal unroll factor for code generation of software pipelining. *International Journal of Parallel Programming*, 41(1):1–58, 2013.

[3] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, A. Betts, J. Ketema, A. F. Donaldson, R. David, and E. Hajiyev. Pencil: a platform-neutral compute intermediate language for accelerator programming. In *under review*, 2015.

[4] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of*

268 *the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages
269 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

270 [5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral
271 parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

272 [6] P. Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*,
273 pages 429–441, St. Malo, France, 1988. ACM.

274 [7] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical
275 tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and*
276 *Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

277 [8] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*,
278 24:649–671, 1998.

279 [9] P. Louis-Noel. PolyBench suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/, 2010.

280 [10] C. Mendis, S. P. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput
281 estimation using deep neural networks. *CoRR*, abs/1808.07412, 2018.

282 [11] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on*
283 *Programming Languages and Systems*, 22(5):773–815, Sept. 2000.

284 [12] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms
285 from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12,
286 July 2012.

287 [13] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjodin, and
288 R. Upadrasta. GRAPHITE two years after: First lessons learned from Real-World polyhedral compilation,
289 Jan. 2010.

290 [14] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-
291 vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*,
292 pages 327–337, 2009.

293 [15] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE*
294 *transactions on parallel and distributed systems*, 2(4):452–471, 1991.

295 [16] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive
296 pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.